Using Java for Reusable Embedded Real-Time Component Libraries

Dr. Kelvin Nilsen
Aonix

Java as a high-level programming language provides great support for a wide variety of networked devices and embedded systems. When used in a military context, Java promises to reduce development and maintenance costs significantly, while increasing reliability, flexibility, and functionality of embedded systems. The secret in Java's success lies in its ability to provide real-time mission-critical response. This article discusses the characteristics of mature Java technologies that are able to meet these important defense criteria.

Originally designed as a language to support "advanced software for a wide variety of networked devices and embedded systems" [1], the Java programming language has much to offer the community of embedded defense system developers. In this context, Java is considered a high-level, general-purpose programming language rather than a special-purpose Web development tool. Java offers many of the same benefits as Ada, while appealing to a much broader audience of developers. The breadth of interest in Java has led to a large third-party market for Java development tools, reusable component libraries, training resources, and consulting services.

Java borrows the familiar syntax of C and C++. Like C++, Java is object-oriented, but it is much simpler than C++ because Java's designers chose not to support compilation of legacy C and C++ code. Due to its simplicity, more programmers are able to master the language. With that mastery, they are more productive and less likely to introduce errors resulting from misunderstanding the programming language.

The Java write-compile-debug cycle is faster than with traditional languages because Java supports both interpreted and just-in-time (JIT) compiled implementations. During development and rapid prototyping, developers save time by using the interpreter. This avoids the time typically required to recompile and relink object files.

Java application software is portable because the Java specification carefully defines a machine-independent intermediate byte-code representation and a robust collection of standard libraries. Byte-code class files can be transferred between heterogeneous network nodes and interpreted or compiled to native code on demand by the local Java run-time environment. The benefits of portability are four-fold:

1. Software engineers can develop and test their embedded software on fast PC workstations with large amounts of memory, and then deploy on smaller, less powerful embedded targets.
2. As embedded products evolve, it is often necessary to port their code from one processor and operating system to another.
3. Cross compiling is no longer necessary. The same executable byte code runs on Power PC, Pentium, MIPS, XScale, and others. This simplifies configuration management.
4. The ability to distribute portable binary software components lays the foundation for a reusable software component industry.

Certain features in Java's run-time environment help to improve software reliability. For example, automatic garbage collection, which describes the process of identifying all objects that are no longer being used by the application and reclaiming their memory, has been shown to reduce the total development effort for a complex system by approximately 40 percent [2]. Garbage collection eliminates dangling pointers and greatly reduces the effort required by developers to prevent memory leaks.

A high percentage of the Computer Emergency Response Team advisories issued every year are a direct result of buffer overflows in system software. Java automatically checks array subscripts to make sure code does not accidentally or maliciously reach beyond the ends of arrays, thereby eliminating this frequently exploited loophole.

The Java compiler and class loader enforce type checking much more strongly than C and C++. This means programmers cannot accidentally or maliciously misuse the bits of a particular variable to masquerade as an unintended value.

Finally as part of the interface definition for components, Java component developers can require that exceptions thrown by their components be caught within the surrounding context. In lower level languages, uncaught exceptions often lead to unpredictable behavior.

Another very useful Java feature is the ability to dynamically load software components into a running Java virtual machine environment. New software downloads serve to patch errors, accommodate evolving communication protocols, and add new capabilities to an existing embedded system. Special security checking is enforced when dynamic libraries are installed to ensure they do not compromise the integrity of the running system.

Though Java programs may be interpreted, it is much more common for Java byte codes to be translated to the target machine language before execution. For many input/output-intensive applications, compiled Java runs as fast as C++. For compute-intensive applications, Java tends to run at one-third to one-half the speed of comparable C code.

Java Within Real-Time Systems

Although Java's initial design was targeted to embedded devices, its first public distributions did not support reliable real-time operation. Several specific issues are identified here and discussed in greater detail in the reference material [3, 4].

Automatic Garbage Collection

Though automatic garbage collection greatly reduces the effort required by software developers to implement reliable and efficient dynamic memory management, typical implementations of automatic garbage collection are incompatible with real-time requirements. In most virtual machine environments, the garbage collector will occasionally put all the application threads to sleep during certain uninterruptible operations while it analyzes the relationships between objects within the heap to determine those no longer in use. The durations of these garbage collection pauses are difficult to predict, and typically vary from half a second to tens of seconds. These problems can be addressed by using a virtual machine that provides real-time garbage collection as described in the following.

Priority Inversion

To guarantee that real-time tasks meet all of their deadlines, real-time developers carefully analyze the resource requirements of each

task and set their priorities according to accepted practices of scheduling theory [5]. Thread priorities are used as a mechanism to implement compliance with deadlines. Unfortunately, many non-real-time operating systems and most Java virtual machine implementations view priorities as heuristic suggestions. This compromises real-time behavior whenever the priorities of certain threads are temporarily boosted in the interest of providing *fair* access to the CPU or to improve overall system throughput. Another problem occurs when low-priority tasks lock resources that are required by high-priority tasks. In all of these cases, the real-time engineer describes the problem as *priority inversion*.

It is important to deploy real-time Java components on virtual machines that honor strict priorities, preventing the operating system from automatically boosting or aging thread priorities, and that build priority inheritance or some other priority inversion avoidance mechanism into the implementation of synchronization locks. Priority inheritance, for example, elevates the priority of a low-priority thread that owns a lock being requested by a high-priority thread so that the low-priority thread can get its work done and release the lock.

Timing Services

Standard Java timing services do not provide the exacting precision required by real-time programmers. Applications that use the Java sleep() service to control a periodic task will drift off schedule because each invocation of sleep() is delayed within its period by the time required to do the periodic computation, and because the duration of each requested sleep() operation only approximates the desired delay time. Also, if a computer user changes the operating system's notion of time while a real-time Java program is running, the real-time threads will become confused because they assume the system clock is an accurate, monotonically increasing time reference. Java virtual machines designed for real-time operation typically provide high-precision and drift-free real-time timers that complement the standard timing utilities.

Low-Level Control

As a modern, high-level programming language, Java's design intentionally precludes developers from directly accessing hardware and device drivers. The ideal is that hardware device drivers should be abstracted by the underlying operating system. However, if Java is up to the task many software engineers would rather do that development in Java than in assembly language or C. Most real-time Java implementations provide services to allow real-time Java components to store and fetch values from input/output ports and memory-mapped devices.

Hard Real-Time Tradeoffs

Developers of hard real-time systems tend to make different tradeoffs than soft real-time developers. Hard real-time software tends to be relatively small, simple, and static. Often, economic considerations demand very high performance and very small footprint of the hard real-time layers of a complex system. To meet these requirements, hard real-time developers generally recognize they must work harder to deliver functionality that could be realized with much less effort if there were no timing constraints, or if all of the timing constraints were soft real-time. Work is under way to define special hard real-time variants of the Java language [6, 7, 8, 9]. One noteworthy difference is that the hard real-time variants generally do not rely on any form of automatic garbage collection.

Real-Time Garbage Collection

One of the most difficult challenges of real-time development with Java is managing the interaction between application code and automatic garbage collection. For reliable operation, there are a number of characteristics that must be satisfied by the garbage collection subsystem. These are described in the following sections.

Preemptive

Typical real-time Java applications are divided into multiple threads, some allocate memory and others manipulate data in previously allocated objects. Both classes of threads may have real-time constraints. Threads that do not allocate memory may have tighter deadlines and run at higher priorities than threads that do allocate memory. Garbage collection generally runs at a priority level between these two classes of priorities. Whenever a higher priority thread needs to run, it must be possible to preempt garbage collection. In some non-real-time virtual machines, once garbage collection begins, it cannot be preempted until it has executed.

Incremental

To assure that garbage collection makes appropriate forward progress, it is necessary to divide the garbage collection effort into many small work increments. Whenever garbage collection is preempted, it must resume with the next increment of work after the preempting task relinquishes control. Real-time garbage collectors avoid the need to restart operations when garbage collection is preempted.

Accurate

We use the term *accurate* to describe a garbage collector that always knows whether a particular memory cell holds a reference (pointer) or holds, for example, numerical representations of integers and floating-point values. In contrast, *conservative* garbage collectors assume memory cells contain pointers whenever there is any uncertainty about the cell's contents. If, interpreted as a pointer, there is an object that would be directly referenced by this pointer, then that object is conservatively treated as live. Because conservative garbage collectors cannot promise to reclaim all dead memory, they are less reliable for long-running mission-critical applications.

Defragmenting

Over the history of a long-running application, it is possible for the pool of free memory to become fragmented. While a fragmented allocation pool may have an abundance of available memory, the free memory is divided into a large number of very small segments, which prevents the system from reliably allocating large objects. It also complicates the allocation of smaller segments because it becomes increasingly important to efficiently pack newly allocated objects into the available free memory segments (to reduce further fragmentation). In general, a real-time virtual machine intended to support reliable, long-running, mission-critical applications must provide some mechanism for defragmentation of the free pool.

Paced

It is not enough to just preempt garbage collection. In large and complex systems, certain activities depend on an ability to allocate new memory to fulfill their real-time-constrained responsibilities. If the memory pool becomes depleted, the real-time tasks that need to allocate memory will necessarily become blocked while garbage collection executes. To prevent this priority inversion from occurring, a real-time virtual machine must pace garbage collection against the rate of memory allocation.

Ideally, the system automatically dedicates to garbage collection activities enough CPU time to recycle dead memory as quickly as the application is allocating memory. However, it does so without dedicating any CPU time that has already been set aside for execution of the real-time application threads, and without dedicating so much CPU time that it completes way ahead of schedule. In a soft or firm real-time system, heuristics are applied to approximate this ideal. The driving considerations are (1) to prevent out-of-memory conditions from

stalling execution of real-time threads, and (2) to maximize garbage collection efficiency by delaying it as long as possible so that each fixed-cost collection reclaims the largest possible amount of dead memory.

Pacing Garbage Collection

Think of garbage collection as a servant to all of the application threads that regularly allocate memory. Because the total garbage collection effort consists of many incremental steps that ultimately benefit all threads that allocate memory, the priority assigned to garbage collection activities must be greater than or equal to that of the highest priority application thread that performs memory allocation.

A pacing agent is the software component responsible for allocating CPU time to the garbage collection effort. In configuring the pacing agent, it is important to identify the maximum priority of threads that allocate memory, the minimum priority assigned to threads having real-time execution constraints, the shortest deadline corresponding to real-time allocating threads, and the percentages of CPU time to be reserved for execution of real-time allocating and non-allocating application threads respectively.

Also, the pacing agent monitors the application to discover behavioral trends, including the rate of memory allocation and the amount of live memory retained following completion of each garbage collection pass. The pacing agent combines all of this information into a coherent approximation of the application's resource requirements. The pacing agent uses this approximation to guide its allocation of CPU time increments to the garbage collection effort.

Among the heuristics applied by the pacing agent are the following:

1. For a given phase of execution, assume future memory allocation behavior resembles previous allocation behavior. An application programming interface service allows the application to inform the pacing agent each time it changes phases. Within a phase, we assume that allocation rates are constant and that retained live-memory is linear in time. This model approximates typical phases such as initialization (during which time large data structures are constructed), steady state execution (during which each new allocation is matched by release of a previously allocated object of similar size), and termination (during which time large data structures may be disassembled).
2. The pacing agent treats garbage collection as a real-time activity with priority at least as high as that of the highest allocating real-time thread. If, however, rate monotonic analysis concludes that there are insufficient CPU resources to guarantee that garbage collection will stay on pace with allocation, the pacing agent endeavors to steal additional CPU cycles for garbage collection at the priority immediately below the lowest priority real-time allocating thread. The pacing agent assures that garbage collection never consumes more CPU time at real-time priorities than would be available according to the rules of rate monotonic analysis [5].
3. To not compromise deadline compliance of real-time allocating threads, the pacing agent takes special care to ensure that its triggering of real-time increments of garbage collection work is periodic with a period no longer than the shortest deadline of the allocating real-time threads. Otherwise, it might end up with the higher priority garbage collection thread having a longer deadline than the lower priority allocating real-time threads, and this would compromise the results of rate monotonic analysis.

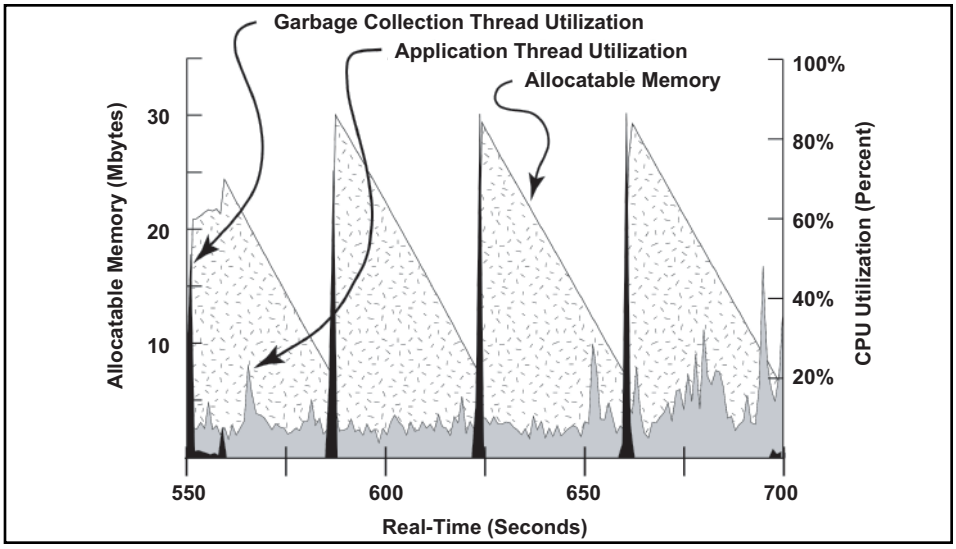
In Figure 1, the amount of allocatable memory is represented by the hashed saw tooth shape, measured according to the scale on the left side of the chart. The amount of CPU time consumed by the simulated air traffic control system's real-time Java application threads is shown in gray. The percentage of CPU time dedicated to real-time garbage collection is illustrated in solid black. The CPU utilization scale is provided on the right-hand side of the chart. These measurements were taken on a computer that was also running a variety of other non-Java tasks. The amount of CPU time taken by the other tasks is not reported directly in this chart. However, the impact of these other activities can be seen indirectly.

For example, the garbage collection effort spanning the time period from roughly 550 to 560 seconds has been preempted by some higher priority activity. During this time span, we see that the garbage collection effort lasts almost 10 seconds at a very low utilization of less than 10 percent following an initial burst of approximately 50 percent CPU utilization. During this same time span, the real-time Java application threads, which happen to run at a higher priority than the garbage collection thread in the measured configuration, are consuming less than 20 percent of the system CPU time.

From these observations, it is clear that the Java virtual machine's garbage collector has been preempted by other higher priority tasks running on the same computer. The ability to mix components written in different programming languages, as demonstrated with this example, is an important capability for mission-critical, real-time Java programming.

This particular application is running in a fairly predictable steady state as characterized by the following observations. First, the slope of the available memory chart is roughly constant whenever garbage collection is idle. This means the application's allocation rate is approximately constant. Second, with exception of the first peak, the heights of the available memory chart's peaks are roughly identical. This means the amount of live memory retained by the application is roughly constant. In other words, the application is allocating new objects at approximately the same rate it is discarding old objects. The reason the first peak is lower than the others is because other high-priority tasks in the system have preempted the garbage collector, delaying its completion. During the time that garbage collection is taking place, new objects con-

Figure 1: *Simulated Air Traffic Control Application With Paced Incremental Garbage Collection*



tinue to be allocated. Therefore, this peak is shorter than the others by roughly the amount of memory that was allocated during the extra wall-clock time required to complete this particular pass of the garbage collector.

Note that the percentage of CPU time consumed by real-time Java application threads is fairly predictable, ranging from about 10 percent to 50 percent, but is by no means constant. This is typical of real-world mission-critical systems. Most real systems exhibit variation in processing requirements as a result of fluctuations in the system workload. In systems that have real-time constraints, resources are budgeted conservatively to make sure there are enough resources to handle the occasional burst of demand for higher processing throughput.

Note that garbage collection is idle most of the time. As memory becomes scarce, garbage collection begins to run. In this example, garbage collection is configured to run at a lower priority than all of the real time application threads. When properly configured, the pacing agent will carefully avoid delaying the application threads by any more than the allowed scheduling jitter even when the garbage collection thread is configured to run at a priority higher than certain real-time Java threads.

Technology Adoption

The soft real-time garbage-collected Java technologies described in this article are commercially available in a cleanroom Java virtual machine product that conforms to the Java 2.0 Standard Edition (J2SE). The technologies have been commercially deployed in a number of mission-critical applications ranging from terabit-per-second fiber-optic switches to soft Programmable Logic Controller control of electric power generation and automation of semiconductor manufacturing. Together, these technology demonstrations represent over 100 developer-years of effort and have produced over a million lines of real-time Java code. Based on their experiences with these projects, developers have consistently found that they are much more productive and their software has fewer errors than when developing with C or C++. Some of their specific experiences are described in [10, 11].

The Nortel Optera HDX long-haul fiber-optic telecommunications switch provides an example of a recent application of real-time mission-critical Java. The hardware architecture for this product is fairly traditional. Redundant shelf controllers are combined in a large air-cooled chassis with a collection of line cards. In this product, both the *line cards* and the shelf controllers are based on PowerPC processors running a commercial real-time operating system.

The line cards have responsibility for the high-performance data transfer operations and implementation of communication protocol stacks. The shelf controllers have responsibility for managing and provisioning the resources contained on the line cards. The high-performance code that runs on the line cards is identified as control plane. The oversight software that runs on the shelf controllers is known as the management plane. Because shelf controllers need to communicate with the line cards, a small amount of management plane software runs on each of the line cards as well.

Previously, Nortel implemented the management plane software in C++; the most recent offering implements this functionality in Java for several reasons:

1. C++ is described as a *big language*, having many complex features that demand highly skilled developers and constant discipline to prevent creeping complexity making it difficult to maintain developed code economically.
2. Dynamic memory management problems in the earlier C++ implementation were particularly troublesome, leading to a variety of memory leaks, dangling pointers, and storage tramplers. Java's automatic garbage collection is to address these issues.
3. The management plane software that runs within the Optera HDX product must communicate with higher-level monitoring and supervisory components running on large Unix servers. In recent years, most of the Network Management System (NMS) and Element Management System (EMS) software running on those Unix servers has been replaced with Java technologies.

In considering whether to use real-time Java for the management plane components, Nortel engineers faced a variety of questions. For example, were the development tools mature enough to support efficient development? Could Java virtual machines run with sufficient reliability to assure the five nines reliability requirements common in the telecommunications industry? Would a real-time Java virtual machine be able to reliably support the 20-ms timing constraints that are imposed on certain management-plane reporting functions? Nortel engineers conducted an extensive yearlong evaluation of available Java technologies before making their final decision to adopt Java for this product.

The task of implementing the Optera HDX management plane software in Java took approximately two years with a development team comprised of more than 40 developers, resulting in a code base of more than a million lines of real-time Java code. Nortel has been selling the Optera HDX product since March 2002. The mission-critical Java components have since proven themselves in many months of successful service in hundreds of commercial deployments.

In evaluating their experience using Java, Nortel engineers have reported the following findings:

- Java software has been more reliable and Java developers have been more productive than their C++ counterparts.
- A large software module developed for the Optera HDX product was easily ported to another hardware platform for a related product.
- The object-oriented discipline utilized with Java made it possible to easily restructure the code to accommodate new requirements midway through development.
- Based on their successes with Java, Nortel intends to use more Java in next-generation products.
- Mistakes made by C programmers occasionally compromise the integrity of the Java virtual machine environment; thus there is motivation to develop lower level high-performance mission-critical Java to complement the soft real-time mission-critical Java that they have already deployed.

Standardization

The standards development being done by the Open Group's Real-time and Embedded Systems Forum [12] will establish a foundation that encourages competitive pricing and innovation among Java technology vendors while assuring portability and interoperability of real-time components written in the Java language. These standards, which are to be endorsed both by the Java Community Process and the International Organization for Standardization, will address a much broader set of requirements than the existing real-time specification for Java.

In particular, the standard for safety-critical Java will address concerns regarding certification under the Federal Aviation Administration's DO-178B guidelines. Beyond requirements for real time, the standard for mission-critical Java will address issues of portability, scalability, performance, memory footprint, abstraction, and encapsulation. Work within the Open Group is ongoing. The current plan is to deliver the safety-critical specification, refer-

ence implementation, and Technology Compatibility Kit by first quarter 2005. Working documents describing the Open Group's Real-time and Embedded Systems Forum's ongoing work standardization activities related to real-time Java are available at <www.opengroup.org/rtforum/rt_java> and <www.opengroup.org/rtforum/rt_safety>.

Table 1 summarizes key differences between different mission-critical Java technologies. The key points emphasized in this table are described in the following bulleted paragraphs:

- The standard J2SE Java libraries are keys to enabling high developer productivity, software portability, and ease of maintenance. Thus, it is important to provide all of these libraries to the soft real-time developer. Unfortunately, the standard J2SE libraries have a significant footprint requirement (at least four megabytes [Mbytes]) and depend heavily on automatic garbage collection, which is not available in the hard real-time environment. Thus, the hard real-time and safety-critical versions of Java cannot use the standard libraries. The hard real-time mission-critical Java standard will support the subset of the Connected Device Configuration libraries that is appropriate for a non-garbage-collected environment running on a limited-service, hard real-time executive. The safety-critical Java standard will support an even smaller library subset, pared down to facilitate safety certification efforts.
- The soft real-time mission-critical Java standard supports real-time garbage collection as described in this article. To improve throughput, determinism, and memory footprint requirements, the hard real-time and safety-critical Java standards do not offer automatic garbage collection.
- In traditional Java and soft real-time mission-critical Java, memory is reclaimed by garbage collection. There is no application programmer interface to allow developers to explicitly release objects, as this would decrease software reliability by introducing the possibility of dangling pointers. In the hard real-time mission-critical environment, we allow developers to explicitly reclaim the memory associated with certain objects. This is a dangerous service that must be used with great care. It is necessary, however, to support a breadth of real-world application requirements. In safety-critical systems, we prohibit manual deallocation of memory as use of this service would make it very difficult to certify safe operation of the software system.
- Traditional Java and soft real-time mission-critical Java allocate all objects in the heap. In the absence of automatic garbage collection, hard real-time and safety-critical Java developers can use special protocols to allocate certain objects on the run-time stack. The protocol includes compile-time enforcement of rules that assure that no pointers to these stack-allocated objects survive beyond the life-time of the objects themselves.
- Dynamic class loading allows new libraries and new application components to be loaded into a virtual machine environment on the fly. This is a very powerful capability, to be provided as broadly as possible. However, current safety certification practices are too restrictive to allow use of this capability in a safety-critical system.
- In the specification for traditional Java, thread priorities are mere suggestions. The virtual machine implementation may honor these suggestions, or it may ignore them. It may, for example, choose to treat all priorities with equal scheduling preference, or it may even choose to give greater scheduling preference to threads running at lower priorities. In all of the real-time Java specifications, priorities are distinct and priority ordering is strictly honored. The safety-critical Java specification implements strict first-in-first-out scheduling within priority levels, with no time slicing. This is the more common expectation for developers of safety-critical systems.
- Traditional Java does not offer any mechanism to avoid priority inversion, which might occur when a low-priority task locks a resource that is subsequently required by a high-priority task for it to make progress. The hard and soft real-time mission-critical specifications both support priority inheritance. Additionally, the hard real-time mission-critical Java standard and the safety-critical Java standard will support the priority ceiling protocol in which particular locks are assigned ceiling priorities, which represent maximum priority of any thread that is allowed to acquire this particular lock. Whenever a thread obtains a lock, its priority is automatical-

	Traditional Java	Mission-Critical Java		
		Soft Real Time	Hard Real Time	Safety Critical
Library Support	J2SE	J2SE	Subset of CDC	Very restrictive subset of CDC
Garbage Collection	Pauses in excess of 10 seconds	Real Time	No garbage collection	
Manual Memory Deallocation	Manual memory deallocation is disallowed		Allows manual deallocation	No manual deallocation
Stack Memory Allocation	No		Safe stack allocation	
Dynamic Class Loading	Yes			No
Thread Priorities	Unpredictable priority clustering and aging	Fixed priority, time-sliced preemptive, with distinct priorities		Fixed priority, distinct priorities, no time slicing
Priority Inversion Avoidance	None	Priority inheritance	Priority inheritance and priority ceiling	Priority ceiling
Asynchronous Transfer of Control	No	Yes		No
Approximate Performance	One-third to two-thirds speed of C	Within 10 percent of traditional Java speed	Within 10 percent of C speed	
Typical Memory Footprint	16+ Mbytes	16+ Mbytes	64 Kbytes - 1 Mbyte	64-128 Kbytes

Table 1: *Proposed Differentiation Between Java Technologies*

ly elevated to the ceiling priority level. If a thread with higher priority than the lock's ceiling priority attempts to acquire that lock, a run-time exception is generated. The priority ceiling mechanism is most efficient and is simpler to implement and to analyze for static systems in which all of the threads and their priorities are known before run time. The priority inheritance mechanism deals better with environments that experience dynamic adjustments to the thread population or to their respective priorities.

- Asynchronous transfer of control allows one thread to interrupt another in order to have that other thread execute a special asynchronous event handler and then either resume the work that had been preempted or abandon its current efforts. This capability, missing from traditional Java, is very useful in many real-time scenarios. We omit this capability from safety-critical systems because the asynchronous behavior is incompatible with accepted practices for safety certification.
- Because of the high-level services supported by Java, including automatic garbage collection, array subscript checking, dynamic class loading, and JIT compilation, traditional Java generally runs quite a bit slower than comparable algorithms implemented in, for example, the C language. Our experience with implementations of soft real-time Java is that they run a bit slower than traditional Java, because real-time garbage collection imposes a greater penalty on typical thread performance than non-real-time garbage collectors. The various compromises represented in the hard real-time and safety-critical Java specifications are designed to enable execution efficiency that is within 10 percent of typical C performance.
- Because of the size of the standard J2SE libraries and a just-in-time compiler, which is present in a typical J2SE deployment, the typical J2SE deployment requires at least 16 Mbytes of memory. Of this total, about half is available for application code and data structures. Depending on the needs of a particular application, the memory requirements may range much higher, up to hundreds of Mbytes for certain applications. Hard real-time mission critical Java is designed specifically to support very efficient deployment of low-level, hard real-time and performance-constrained software components. Though different applications exhibit different memory requirements, targeted applications typically run from about 64 kilobytes (Kbytes) up to a full Mbyte in memory requirements. Safety-critical deployments tend to be even smaller. This is because the costs of certification are so high per line of code that there is strong incentive to keep safety-critical systems as small as possible.

Conclusions

Increasingly, the military relies upon intelligence implemented as real-time software components to give their warfighters competitive advantage in modern conflicts. Developing and maintaining these large software systems represents tremendous cost and a high degree of risk. High-level programming languages like Java promise to reduce development and maintenance costs by two- to 10-fold, while increasing the reliability, flexibility, and functionality of embedded real-time systems.

Though early implementations of the Java virtual machine failed to address the needs of mission-critical real-time developers, newer technologies bring the full benefits of Java to this very important defense community.◆

References

1. Gosling, J., and H. McGilton. "The Java Language Environment: A White Paper." Mountain View, CA: Sun Microsystems, Inc., May 1996 <http://java.sun.com/docs/white/ langenv>.
2. Rovner, P. "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked Concurrent Language." Palo Alto, CA: Xerox Palo Alto Research Center, 1984 <www.parc.xerox.com/about/history/pub-historical.html>.
3. Nilsen, K. "Issues in the Design and Implementation of Real-Time Java." Real-Time Magazine Mar. 1998 <www.realtime-info.be/magazine/98q1/1998q1_p009.pdf>.
4. Nilsen, K. "Adding Real-Time Capabilities to the Java Programming Language." Communications of the ACM 41.6 (June 1998): 49-56 <http://doi.acm.org/10.1145/276609.276619>.
5. Klein, M., T. Ralya, B. Pollak, and R. Obenza. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Nov. 1993.
6. Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The Real-Time Specification for Java. Addison-Wesley, Jan. 2000.
7. J. Consortium's Real-Time Java Working Group, et al. "Real-Time Core Extensions." Cupertino, CA: J. Consortium. 2 Sept. 2000 <www.j-consortium.org/rtjwg/rtce.1.0.14. pdf>.
8. Nilsen, K., and A. Klein. Issues in the Design and Implementation of Efficient Interfaces Between Hard and Soft Real-Time Java Components. Proc. of the Workshop on Java Technologies for Real-Time and Embedded Systems. Catania, Sicily, Italy, 3-7 Nov. 2003.
9. Nilsen, K. Doing Firm Real-Time With J2SE APIs. Proc. of the Workshop on Java Technologies for Real-Time and Embedded Systems. Catania, Sicily, Italy, 3-7 Nov. 2003.
10. NewMonics, Inc. "Calix Success Story." Tucson, AZ: NewMonics, Inc., May 2003 <www.newmonics.com/perceval/calix_whitepaper. shtml>.
11. NewMonics, Inc. "Nortel Success Story." Tucson, AZ: NewMonics, Inc., Oct. 2003 <www.newmonics.com/perceval/nortel_whitepaper. shtml>.
12. The Open Group. Real-time and Embedded Systems Forum <www.opengroup.org/rtforum>.

About the Author

Kelvin Nilsen, Ph.D., is chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including Ameos MDA tools; ObjectAda compilers, development environment, and libraries; SmartKernel runtime executives; and commercial off-the-shelf safety certification support. Nilsen's pioneering research in real-time programming resulted in seven commercial patents. His seminal research on the topic of real-time Java led to the founding of NewMonics, a leader in advanced clean-room Java technologies. In 2003, Aonix acquired NewMonics. Nilsen has a Bachelor of Science in physics from Brigham Young University and a Master of Science and doctorate degree both in computer science from the University of Arizona.

Aonix
877 S. Alvernon WY STE 100
Tucson, AZ 85711
Phone: (520) 323-9011
Fax: (520) 323-9014
E-mail: kelvin@aonix.com